Gene Homology Explorer

Final Report

https://github.com/jogoodma/homology-explorer

Josh Goodman & Mark Green

Spring 2023

Contents

1 Abstract **1**

2 Introduction **2**
- 2.1 Problem Statement 2

3 Methods **3**
- 3.1 Homology Data 3
- 3.2 Web Application Design 5
- 3.3 Proxy Server 7
- 3.4 Application Programming Interface 7
  - 3.4.1 Stage 1: Build Database 7
  - 3.4.2 Stage 2: Build Gene Network API 8
- 3.5 User Interface 12
  - 3.5.1 Core Dependencies 12
  - 3.5.2 Sigma.js 12
  - 3.5.3 Dynamic Search 12
  - 3.5.4 Visualization 12

4 Results **12**
- 4.1 PTEN 14
- 4.2 BCL6 14
- 4.3 18w 16

5 Discussion **16**
- 5.1 Outcomes 18
- 5.2 Limitations 18
- 5.3 Next Steps 19

6 Acknowledgments **19**

7 References **19**

1 Abstract

Genetic homology between species is an important tool that researchers rely on to study human disease models in other species, gene function, metabolic pathways, the impact of genetic variants, and more. Existing tools to predict and model these relationships do not provide researchers a wholistic view of genetic homology, are difficult to efficiently navigate, or are lacking in the species or algorithms offered. The Gene Homology Explorer project aims to remedy this by creating, visualizing, and analyzing the network of homological gene relationships between species with a web application. Herein, we describe

the data sources, data processing, modeling, network analysis methods, and application design used to address these needs. To evaluate the utility of the tool, we discuss three example cases that highlight interesting insights that this new application provides. Finally, we conclude with a summary of our project outcomes, current limitations of the Homology Explorer tool, and possible next steps for future work.

2 Introduction

In the field of human genetic research, model organisms play a crucial role in helping to decipher functional mechanisms, disease mechanisms, variant impact, gene therapy, and many other aspects of genes (Millburn, GH, et. al; Bulaklak, K, et. al). Researchers in this field of study rely on previously published data in their organism of interest and also related organisms to discover as much information as possible. For example, a geneticist studying the KRAS gene in humans might look for studies on related genes in mice or rats before designing experiments or looking for drug targets. These related genes are called *orthologs*. Orthologs are homologous genes that are the result of a speciation event (Koonin EV, et. al). In other words, a gene in one species that is directly, but possibly distantly, related to a gene in another species over an evolutionary time period. *Paralogs*, genes that are the result of a duplication event within a species, can also be used for this same purpose.

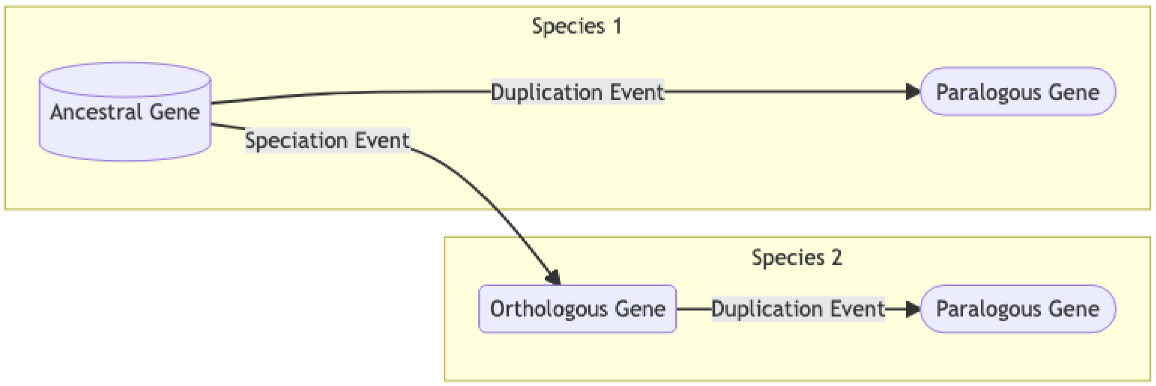


Diagram 1 - Flowchart illustrating the difference between orthologous and paralogous genetic relationships as the result of evolutionary speciation and duplication events.

Although there are many methods for predicting homologous relationships between genes, recent related works (Hu et. al) have compiled a unified database cataloguing these relationships. While this is extremely useful for researchers, the tabular format of the database obscures the network structure inherent in the data - whereby edges are predictions of relationships between genes, and the genes themselves are nodes.

This research aims to close the gap between biomedical researchers and the gene homology information by visualizing the gene homology network as an interactive, searchable web application capable of exploring orthologous and paralogous relationships of individual genes, of lists multiple genes, and of their respective gene neighborhoods. To enhance the researchers understanding of these relationships, we also aim to provide network analysis tools via the UI to perform network operations of centrality measurement and link community detection.

2.1 Problem Statement

Although data have been compiled to catalogue and rank relationships between genes predicted by a variety of different models, there is not an easily accessible way for researchers to access this data. Previously published attempts at providing a solution have either focused on a very limited set of species (Tulpan et. al) or a limited set of methods for determining homology (Mustafin et. al; Nevers et. al). This leads us to the first and main problem:

Provide an easily accessible visual interface allowing model organism genetic researchers to intuitively explore the gene homology network.

To solve this problem, the team has decided to design a web application to serve network visualization and analysis of the network data in a way that allows users to explore the gene homology network with focus and ease.

Some additional problem constraints were as follows:

- The application should be easily hostable, so that it can be run turn-key and independently by researchers or hosted on the web.
- The application should be designed to a production-grade standard such that the application could support a moderate amount of concurrent users.
- The application should provide an interactive and reactive user interface and allow for users to both search for and drill into various different genes and their surrounding neighborhoods.
- Users should be able to easily execute a variety of network analysis methods and graph configurations while maintaining the ability to constrain their results through parameterization of the various network operations.

3 Methods

The `Homology Explorer` application is an answer to the problem statement above. The sections below outline the technical methodology applied to design and build a prototype of this application.

3.1 Homology Data

The source of the homology network data consist of three static `.tsv` tables that were obtained from the DIOPT research group (Hu et. al). These static files contain the unified orthology and paralogy calls and meta information from over 17 homology prediction algorithms.

Orthology and paralogy calls for the following model organism species were included in the Homology Explorer tool from the DIOPT dataset (v8.5).

Model Organism Species - Arabidopsis thaliana (Thale cress) - Schizosaccharomyces pombe (Fission yeast) - Saccharomyces cerevisiae (Yeast) - Caenorhabditis elegans (Worm) - Drosophila melanogaster (Fly) - Danio rerio (Zebrafish) - Xenopus tropicalis (Western clawed frog) - Rattus norvegicus (Rat) - Mus musculus (Mouse) - Homo sapiens (Human)

The table below outlines the attributes for each table listing the datatype to the left of the attribute name and either `pk` or `fk` to designate fields that are primary keys (to uniquely identify records) and foreign keys (to relate attributes across the tables).

Some notes on each table and its attributes:

- `OrthologPairs.tsv` :: 11,099,012 records
  - This table contains the *edge* data for the network.
  - For a given directed edge uniquely identified by `opb_id`, `geneid1` and `geneid2` correspond to the source and target nodes and the `score` corresponds to the edge weight.
  - The `score` attribute denotes the number of prediction algorithms that predict an homologous relationship from a source gene to a target gene. `best_score` indicates whether a given directed edge has superior weight compared to its inverse edge (directed instead from the *target* to the *source*). `confidence` is a category derived from `score`.
- `GeneInfo.tsv` :: 371,760 records
  - This table contains the *node* data of attributes associated with a given gene.
  - The `GeneInfo.geneid` field relates to the `OrthologPairs.geneid*` while the `symbol` and `description` help to colloquially identify a given gene.
  - `chromosome` and `gene_type` are additional descriptive attributes for the gene, while the `map_location` is a legacy methodology for identifying different genes.
- `Species.tsv` :: 10 records

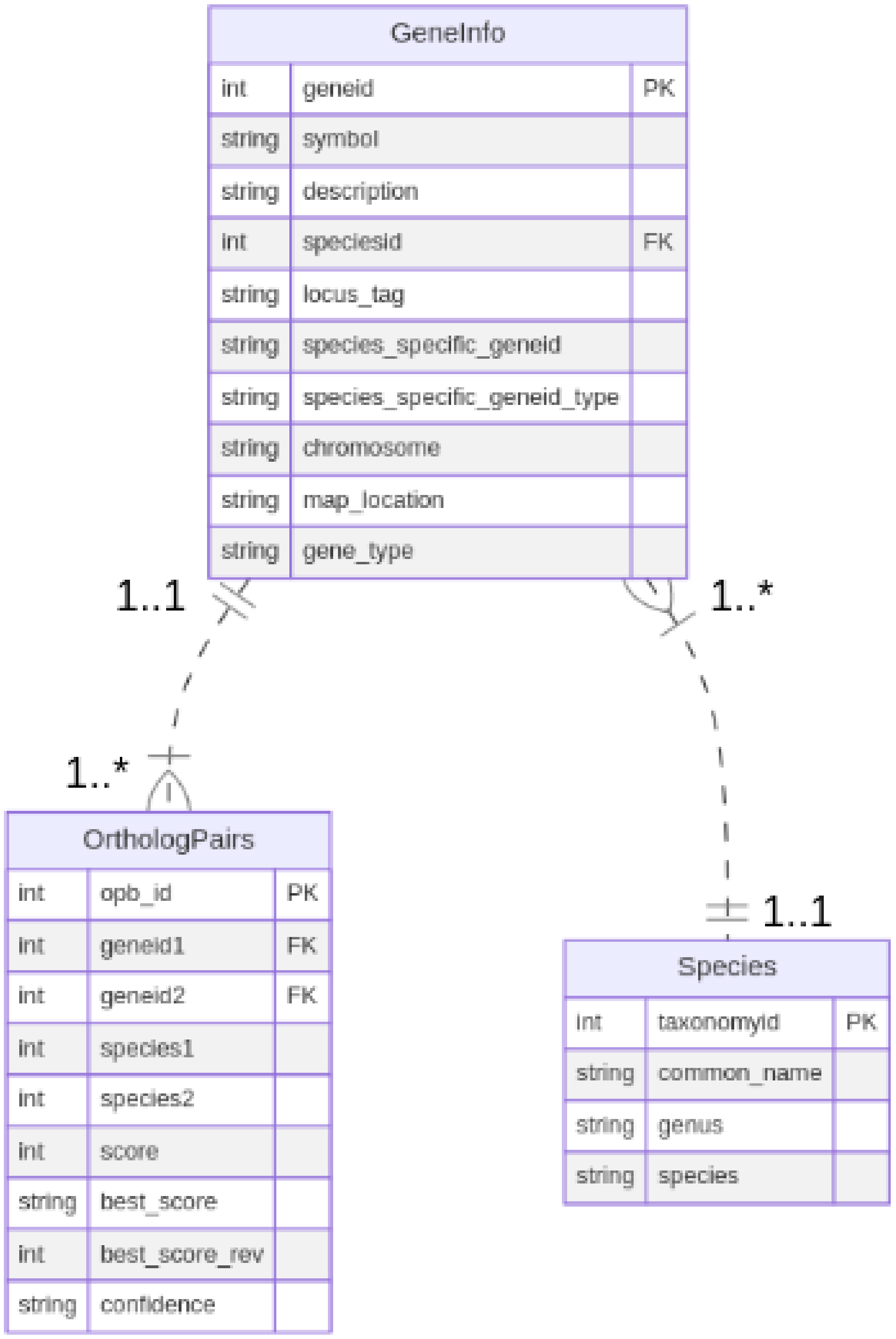


Figure 1: Entity Relationship Diagram of the source data tables

  - This table contains taxonomy information for the species to which the genes in our database belong.
  - The `Species.taxonomyid` field relates to the `GeneInfo.speciesid` field to provide a given species' common and scientific/latin names.

3.2 Web Application Design

At its core, the `Homology Explorer` application aims to serve network data visualizations to users. An application is designed consisting of four key components to accomplish this task, as overviewed below:

- **Pre-processing the source data** converts the `.tsv` files into a `DuckDB`[1] database format. This stage joins and permutes the data to a form that is embeddable as a volume onto the web application.
- A `Caddy`[2] **Proxy Server** manages SSL encryption and routes HTTP traffic between the API and UI services, and the users.
- The backend **Application Programming Interface** (API) service accesses the database through a `SQLAlchemy`[3] object relational mapping (ORM) and implements the python `FastAPI`[4] framework to aynchronously manage database access, validate, serialize, and apply CRUD (Create Read Update Delete) operations and `Networkx`[5] network analytics to the data.
- The frontend **User Interface** (UI) service runs the visualization application by implementing the `Sigma.js`[6] and `Graphology`[7] network libraries through a javascript `React`[8] framework.

The schematic below outlines the described web application architecture. The Homology Explorer is containerized as a Docker Compose[9] application made up of three Docker[10] containers for the API, UI, and Proxy services. The application can be run by executing the command `docker compose up` from the `homology-explorer` root directory.

[1] https://duckdb.org/why_duckdb
[2] https://caddyserver.com/docs/
[3] https://www.sqlalchemy.org/
[4] https://fastapi.tiangolo.com/
[5] https://networkx.org/
[6] https://www.sigmajs.org/
[7] https://graphology.github.io/
[8] https://react.dev/
[9] https://docs.docker.com/compose/
[10] https://docs.docker.com/

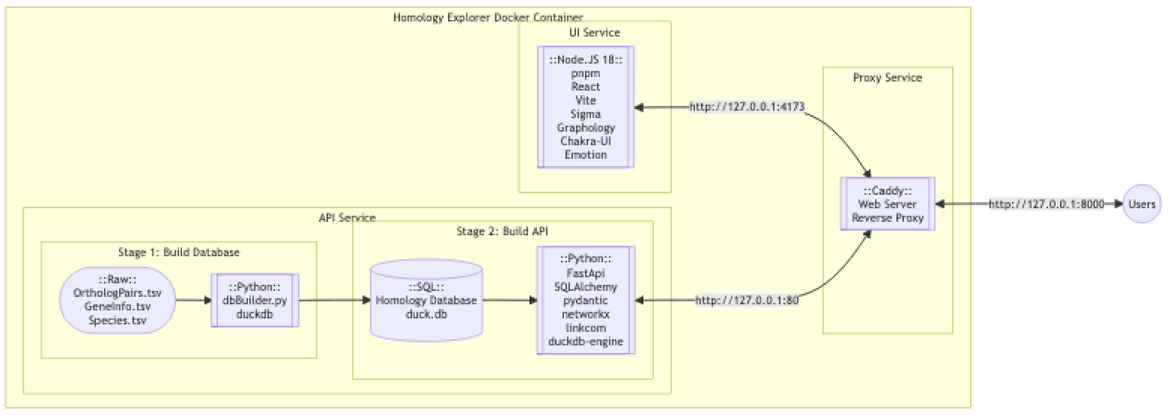


Diagram 2 - Docker compose application consisting of containers running services for the database build script and API, the UI, and the Proxy server.

```
homology-explorer
*
|-- apps
|   |-- api
|   |   |-- app
|   |   |-- Dockerfile
|   |
|   |-- ui
|       |-- node_modules
|       |-- public
|       |-- src
|       |-- Dockerfile
|
|-- data
|-- docs
|-- Caddyfile
|-- docker-compose.yml
|-- pnpm-lock.yaml
|-- poetry.lock
|-- pyproject.toml
```

The directory tree above outlines the general organizational structure of the Homology Explorer application and will be revisited in greater detail throughout the report. The code for this project is available on Github[11]. The subsequent sections outline the technical implementation of the above described web application design in such a way as to satisfy the constraints set out by the problem

[11]https://github.com/jogoodma/homology-explorer/tree/main

statement.

3.3 Proxy Server

A proxy server is an application which controls traffic both within networked systems and between other applications and the internet. These services are often embedded within larger applications and serve multiple other roles like encryption, authentication, cacheing, and load balancing.

The Homology Explorer uses a simple out-of-the-box proxy server solution called `Caddy` because it is easy to implement on small projects without many modifications and is modestly secure with the default settings. Since web hosting is not the feature of this report, this section will be left deliberately sparse for brevity.

3.4 Application Programming Interface

An application programming interface (API) can be used to efficiently broker, transform, and validate data between the database and the UI by managing the database requests made by the UI as the user interacts with the application. The API Docker container has a multi-stage build process:

1. Stage 1 parses the provided `.tsv` files into the `duck.db` format and loads it as a container volume.
2. Stage 2 runs the API service connected to the database volume.

3.4.1 Stage 1: Build Database

The source data have a fixed schema and are relational so managing them through a SQL database is a natural design choice. Of the myriad flavors of SQL to choose from, the Homology Explorer uses `DuckDB`. `DuckDB` is a lightweight SQL database similar to SQLite. Because the dataset however, it is more performant and provides many more features than SQLite. It also has good documentation and a rich python API. To construct the database, these tables are first extracted from the `tar.gz` file. Next the `duckdb` python library is used by `dbBuilder.py` file to create the application database: `duck.db`.

```
data
*
|-- GeneInfo.tsv
|-- OrthologPairs.tsv
|-- Species.tsv
|-- dbBuilder.py
|-- duck.db
```

Each `.tsv` file is loaded to the database as a table. However, the tables themselves are not exposed to the web application. Instead, a variety of views query data from the tables into formats expected by the API object relational mappings. This is advantageous from an application performance standpoint since initializing the data into the ORM formats sometimes requires computationally intensive join operations for which SQL is optimized.

Below are described the various tables and views created by the `dbBuilder.py`.

3.4.1.1 Tables .

- **tblOrthologPairs**
  - Original table of attributes for each `opb_id` (edge).
- **tblGeneInfo**
  - Original table of attributes for each `geneid` (node).
- **tblSpecies**
  - Original table relating `species_id` to common and scientific names.

3.4.1.2 Views .

- **evwGeneFrequency**
  - View aggregates `tblOrthologPairs` to count the frequency of each `geneid`.

- **evwSymbolSearch**
  - View joins to `tblGeneInfo` the `tblSpecies.common_name` and `evwGeneFrequency.frequency` for queries related to UI dynamic search results.
- **evwGeneInfo**
  - View joins to `tblGeneInfo` the `tblSpecies` attributes to provide a enrich queries accessing node attributes.
- **evwOrthologPairs**
  - View adds edge attribute `homolog_type` as binary classification of a given homolog as either an *ortholog* (inter-species) or a *paralog* (intra-species).
- **evwGeneNeighborEdges** & **evwGeneNeighborEdgesAttr**
  - Views are called together to serialize the edge attributes from `tblOrthologPairs` as a nested dictionary below the `key`, `source`, and `target`.
  - These views *only* select edges which are the `best_score`.
- **evwGeneNeighborEdgelist**
  - View called to serialize the edge attributes from `tblOrthologPairs` into an edgelist format readable by `networkx`.
- **evwGeneNeighborNodes** & **evwGeneNeighborNodesAttr**
  - Views are called together to serialize the node attributes from `tblGeneInfo` as a nested dictionary below the `key`.
  - The `evwGeneNeighborNodesAttr` view also joins the `tblSpecies` attributes to `tblGeneInfo` to enrich the attribute output.

The `dbBuilder.py` script constructs this `duck.db` database using the DuckDB python API. Once constructed, it is then embedded the API service at `apps/api/app/duck.db` where it is accessed by the web application as needed.

3.4.2 Stage 2: Build Gene Network API

Python is the preferred language for this API because the application relies on the extensive `networkx` python library to compute network analyses. Keeping the API semantics within the same language as the main computational library maintains a level of desirable level of simplicity. The second stage of the API Docker container build implements the python `FastAPI` framework the API service for the Homology Explorer.

`FastAPI` is a popular python RESTful[12] API web framework focused on high performance and quick development. We selected the framework for our web application backend because of these features which the Homology Explorer leverages:

- the API can run asynchronously through an Asynchronous Server Gateway Interface[13] (ASGI) like `Uvicorn`[14] which allows it to efficiently enqueue, await, and handle API calls from multiple concurrent users. It also boosts performance for API calls requiring multiple SQL queries to construct the response body.
- it supports *object relational mapping*, a method of mapping objects in an object-oriented framework to relational objects in SQL, through the `SQLAlchemy` library to mix the database queries seamlessly with the `networkx` python library.
- `FastAPI` supports datatype validation through the `types` and `pydantic`[15] libraries. This ensures the API is sending an appropriate valid response to a requester, and helps determine if the request is even valid in the first place.
- `FastAPI` has the Swagger Docs interface tool enabled by default which makes developing the API much easier by providing an easy way to test different API requests, responses, and ORM schemas.
- There is good documentation and it is a relatively easy framework to learn and implement. For example, this API application is largely inspired by this official tutorial[16] from the `FastAPI` documentation.

[12]https://restfulapi.net/
[13]https://asgi.readthedocs.io/en/latest/
[14]https://www.uvicorn.org/
[15]https://pydantic.dev/
[16]https://fastapi.tiangolo.com/tutorial/sql-databases/

`FastAPI` also supports several methods for implementing a cache-aside framework, which can help increase performance for high I/O applications, or parallelization of computationally intensive tasks. Since these are not currently part of an expected use-case scenario for this application, this version has *not* been developed with these features. However, its worth noting these may be advantageous features one day in the future if the application use-case ever expands to serve a larger audience (cacheing) or users want to perform analyses on larger and larger graphs (parallelization).

Below is an example directory structure of the `FastAPI` API service and a summary of the different file roles.

```
apps/api
*
|-- app
|   |-- __init__.py
|   |-- crud.py
|   |-- database.py
|   |-- main.py
|   |-- models.py
|   |-- schemas.py
|
|-- Dockerfile
|-- requirements.txt
```

- **database.py**
  - This file handles ORM session connections to the database with `SQLAlchemy`.
- **models.py**
  - This file structures the SQL tables into an object relational mapping with `SQLAlchemy`.
  - Once instantiated as an ORM object, the tables are easy to manipulate using other python libraries.
- **schemas.py**
  - This file enforces data types and structural representations of ORM objects with `pydantic`.
  - Both objects used for posts and responses are validated according to these schema.
- **crud.py**
  - This file parses, analyzes, and transforms ORM objects with `SQLAlchemy` and `networkx`.
- **main.py**
  - This file functionalizes the `crud.py` operations and serves the requests through HTTP API calls with `FastAPI`
  - It also validates POST and GET operations and constrains inputs/outputs according to design specifications, and assigns query and path parameters to HTTP routes.

This API is run through the `Uvicorn` ASGI as recommended by the `FastAPI` documentation and works more or less as follows for each of the API endpoints:

- The UI sends an HTTP request with the query and path parameters specified by one of the available API endpoints in `main.py`.
- If a valid endpoint is called, the endpoint then:
  - **Validates the *Request*** body schema and the query and path parameter datatypes.
  - **Executes** CRUD operations by transforming the ORM models connected to the database.
  - **Serializes** a response to a particular output format.
  - **Validates the *Response*** datatypes according to a given output schema.
  - **Returns** the HTTP response body to the requester.

And that's it! The API endpoints all vary somewhat depending on their intended purpose, but follow this generalized order of operations. Some API endpoints only involve one database query while others are composed of several database queries. In the case of this API, all of the endpoints are designed to help a user explore the network of genes stored by the gene homology database. These endpoints are discussed individually at length in the sections below.

3.4.2.1 Information and Search .

GET Calls:

- `/search/gene/symbol/{symbol}/`
- `/geneinfo/gene/{gene_id}/`
- `/orthologpairs/gene/{gene_id}/`

These endpoints query basic information from views of the database tables. The `/search/gene/symbol/` endpoint is used to populate search results for the dynamic search option at the beginning of the UI. The other two endpoints return either the node attributes or a list of edges attached to the single requested gene.

3.4.2.2 Multigene Requests .

POST Calls:

- `/geneinfo/multigene/`
- `/orthologpairs/multigene/`

These endpoints require the user to post the request body as a list of `geneids`. The endpoint parses the list by changing the comparative of the SQL `WHERE` clause from `=` (as used in the `/.../gene/` GET endpoints) to `IN`. In this way, the multigene endpoints are able to provide node and edge info in a similar manner to the single gene endpoints, but for a list of genes instead.

3.4.2.3 The Gene Neighborhood .

GET Calls:

- `/geneneighboredges/gene/{gene_id}/`
- `/geneneighbornodes/gene/{gene_id}/`

POST Calls:

- `/geneneighboredges/multigene/`
- `/geneneighbornodes/multigene/`

These `geneneighbor*` endpoints are composed of two CRUD operations. The first CRUD operation for each endpoint computes the *gene neighborhood*, a list of all the distinct first degree neighbors of the requested gene or genes. This operation can be parameterized by an upper or lower bound to constrain the neighbors to only those first degree neighbors whose weights are constrained by the bounded interval. This provides a way to filter a neighborhood's edges by weight.

By calling a commonly parameterized initialization function the gene neighborhood allows for multiple different API calls to reference the same subgraph of nodes. This makes these kinds of calls easier for developers to implement by allowing the same subgraph to be easily re-queried for a variety of different information.

The second CRUD operation then fetches either: 1. Nodes - and then returns gene information for each node in the gene neighborhood of the requested gene or genes; or, 2. Edges - and then returns a list of homolog edges for each edge between each node pair within the gene neighborhood of the requested gene or genes.

The gene neighborhood represents the subgraph composed of all the edges between all the first-degree neighbors of a queried gene or genes. The API then returns either the list of nodes or edges associated with the subgraph.

3.4.2.4 Network Analysis .

POST Calls:

- `/geneneighboredges/multigene/{analysis}`
- `/geneneighbornodes/multigene/{analysis}`

A network analysis endpoint is available for computing network attributes on either nodes or edges while the requested network analysis is passed to the endpoint as a path parameter string. Additional query parameters can also be used to toggle the output of the analysis by tuning various hyperparameters. Network analysis endpoints are constructed in a manner similar to the multigene gene neighborhood endpoints and operate as follows:

1. The MultiGene Neighborhood query is called and the resultant edge information are serialized to an edgelist format - a list of 3-dimensional tuples composed of the source and target genes and weight for a given edge.
2. The edgelist is converted to an `nx.Graph()` object upon which is run a `networkx` function of some kind. This returns an edgelist or nodelist with the newly calculated network analysis attribute.
3. The newly calculated attribute from step 2 is appended to the node or edge attributes for the gene neighborhood determined in step 1. This returns a new list of edges or nodes along with all the attributes (original and calculated) for each object of the list.

This framework is advantageous for later development since the only difference between each network analysis is the CRUD operation performing the `networkx` function called in step 2. These CRUD operations are quite simple to construct, and then it is just a matter of making the path parameter available as a request to the network analysis endpoint.

For example, the network CRUD operation for the `geneneighbornodes/multigene/PageRank/` call only consists of 8 lines:

```python
def get_Pagerank(db: Session, edgelist: list, alpha: float):

    g = nx.DiGraph()
    g.add_edges_from(edgelist)

    pagerank = nx.pagerank(g, alpha=alpha, weight='weight')

    newnodeattr = [
        { 'key': key, 'attributes': { 'pagerank': value } }
        for key, value in pagerank.items()
    ]

    return newnodeattr
```

... and the network CRUD operation for the `geneneighboredges/multigene/Linkcom/` call consists of scant more:

```python
def get_Linkcom(db: Session, edgelist: list, threshold: float):

    g = nx.Graph()
    g.add_edges_from(edgelist)

    e2c,_ = linkcom.cluster(
        g, threshold=threshold,
        is_weighted=True, weight_key='weight',
        to_file=False
    )

    newedgeattr = [
        {
          'source': key[0],
          'target': key[1],
          'attributes': {
            'linkcom': value
          }
        } for key, value in e2c.items()
    ]
    return newedgeattr
```

3.5 User Interface

The final piece of the Homology Explorer application is the user interface (UI). The UI serves as the interface between the end user exploring the network data and the API and database layer. Our goal was to design a tool that addressed the points discussed previously in our *Problem Statement*.

3.5.1 Core Dependencies

The user interface for the Homology Explorer is primarily written in `Typescript`[17] with the following libraries providing the core functionality:

- `React`[18] - View layer responsible for controlling the overall layout, the graph components, user interactivity, and managing application state
- `react-sigma`[19] - Utility library for bridging functionality between `React` and `Sigma.js`
- `Sigma.js`[20] - Graph visualization library
- `Graphology.js`[21] - Graph library for representing and working with graph data structures.

A list of all dependencies can be found in the UI package.json file[22].

3.5.2 Sigma.js

Sigma.js was chosen over other web based network visualization libraries (e.g. CytoscapeJS and Vega) for its use of Canvas vs SVG and its tight integration with Graphology (a JS graph library). Canvas was preferred because it provides better performance over SVG when the number of objects displayed is large.

3.5.3 Dynamic Search

The landing page of the Homology Explorer is our simple dynamic search page. This page has a single input field that allows a user to enter a gene symbol. As the user types, a REST query is sent to the gene symbol query API endpoint (refer to “GET Calls” under *Information and Search*), an autocomplete box appears, and then the results from the query are displayed (Figure 2). Clicking the “View Network” box then takes the user to the primary network visualization.

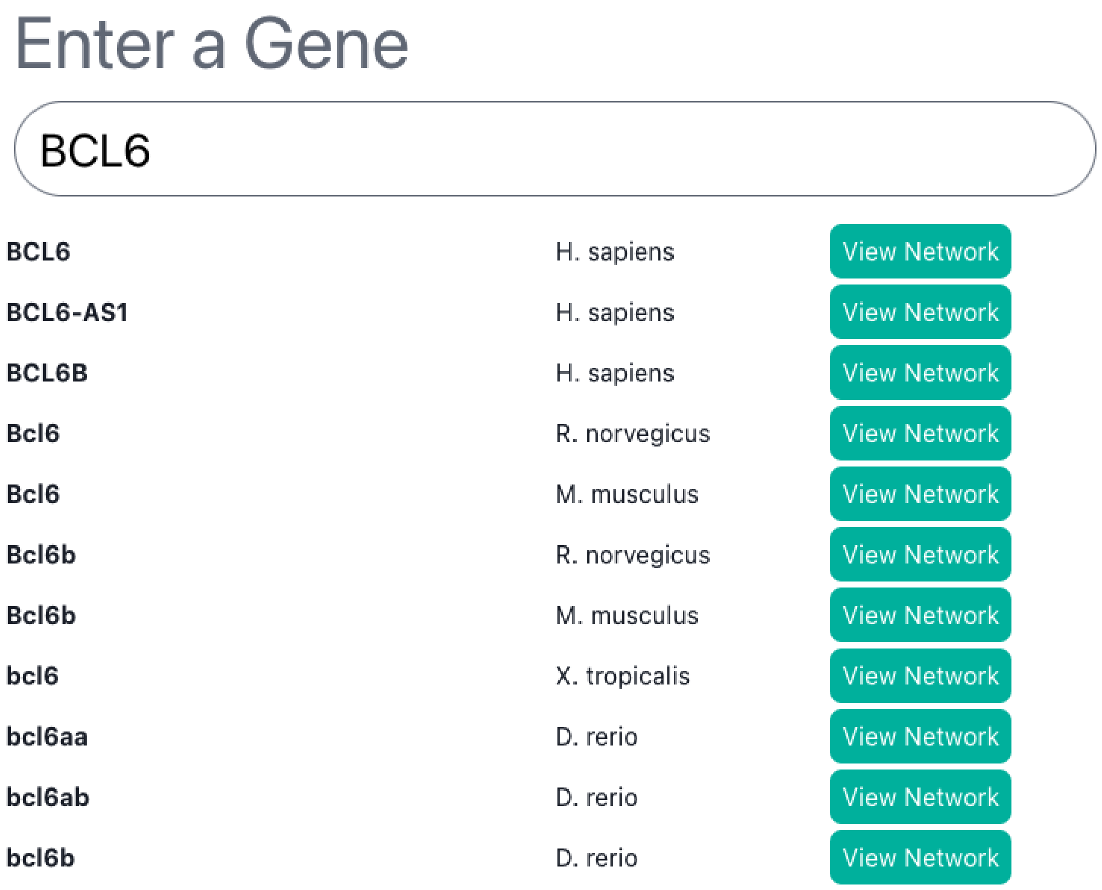
3.5.4 Visualization

The network visualization is controlled by `React` components that manage the application views and state. The `GeneNetwork` component takes the requested gene ID, the application state, event handlers, and creates the network visualization via the `react-sigma` bindings for the `Sigma.js` library. The `Graphology.js` library was used to implement the some interactivity and graph filtering logic provided by the application.

4 Results

In this section, we discuss three examples that demonstrate how the database, model, API, and view layers all come together to produce a network visualization tool that addresses our primary goals. Each example, shows a network for a single gene and what information is learned thanks to the interactive features of the tool, the network analysis algorithms, and the data sources selected for this project.

[17]https://www.typescriptlang.org/
[18]https://react.dev
[19]https://sim51.github.io/react-sigma/
[20]https://www.sigmajs.org
[21]https://graphology.github.io/
[22]https://github.com/jogoodma/homology-explorer/blob/docker/apps/ui/package.json



Enter a Gene

BCL6

BCL6	H. sapiens	View Network
BCL6-AS1	H. sapiens	View Network
BCL6B	H. sapiens	View Network
Bcl6	R. norvegicus	View Network
Bcl6	M. musculus	View Network
Bcl6b	R. norvegicus	View Network
Bcl6b	M. musculus	View Network
bcl6	X. tropicalis	View Network
bcl6aa	D. rerio	View Network
bcl6ab	D. rerio	View Network
bcl6b	D. rerio	View Network

Figure 2: Example of a dynamic search for BCL6

4.1 PTEN

In Figure 3, the gene homology network for the Human PTEN gene is shown. PTEN is a tumor suppressor that when mutated results in a wide variety of cancers (PTEN, Alliance of Genome Resources). PTEN shows clear orthology to genes in other species (light orange edge colors) and a paralog TPTE (highlighted). The paralog TPTE, shows strong homology to a cluster (purple edge color) of model organism genes and another human gene (TPTE2). There are also 5 other predominate link communities that are indicated. In this example, we show that a connection that is 2-3 degrees away is clearly visible in a single visualization with the Linkcom analysis. Compared with the tabular view, this would have required several additional clicks and sorting through dozens of genes.

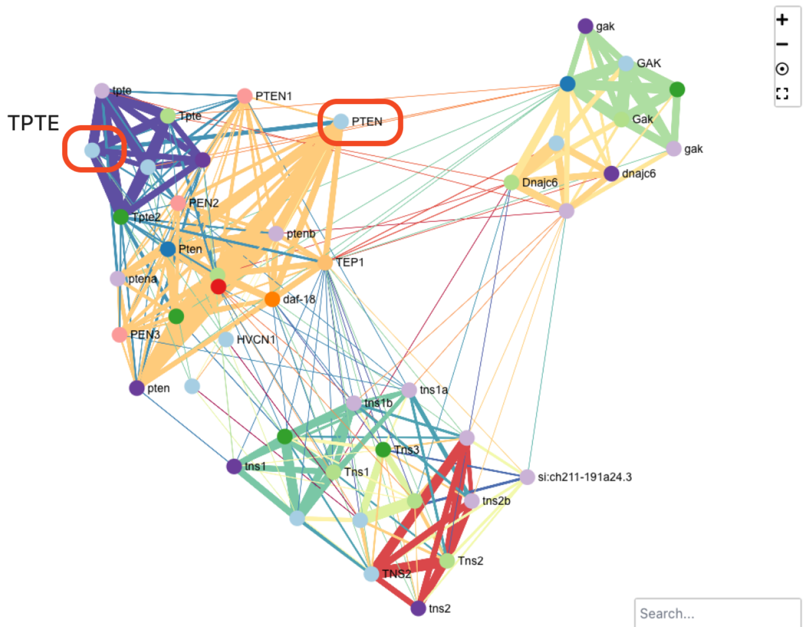


Figure 3: Gene homology network of PTEN. PTEN and its human paralog (TPTE) are highlighted in red.

4.2 BCL6

In Figure 4, the gene homology network for the Human BCL6 gene is shown. BCL6 is a gene known to be involved in B-cell lymphoma (BCL6, Alliance of Genome resources). BCL6 shows homology to a small cluster of genes with high confidence as indicated by the thick edges of blue and cyan and low confidence orthology calls to 4 *D. melanogaster* genes. The node size of those genes has been accentuated by the PageRank centrality analysis (highlighted in red). Additionally, these 4 genes all have symbols with CG#, indicating that their function is not known or that they are not well studied. The graph also indicates that these genes have many connections to a cluster of human genes (yellow link edges) that are all zinc finger type genes. This information could be used for additional experimentation to confirm zinc finger activity of the 4 *D. melanogaster* genes.

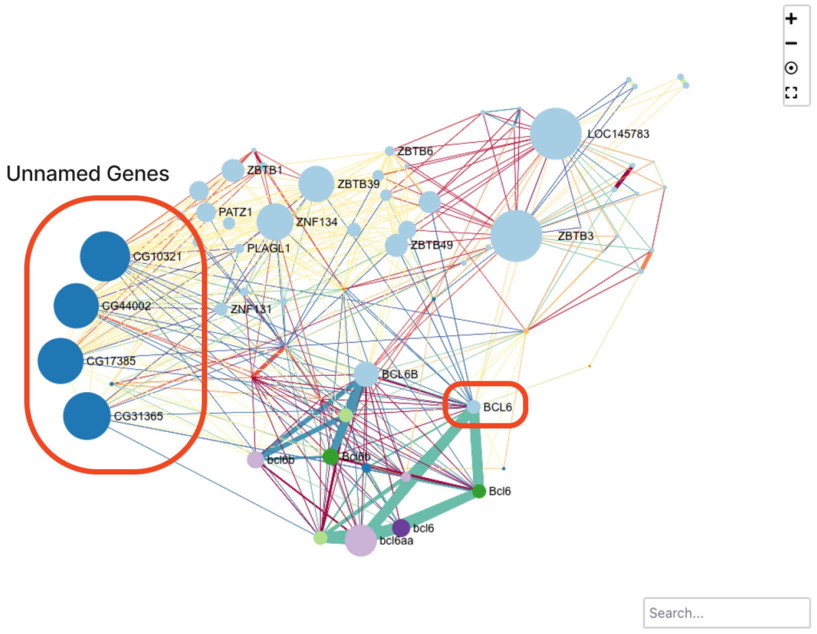


Figure 4: Gene homology network of BCL6. BCL6 and a cluster of 4 *D. melanogaster* genes are highlighted in red. The unnamed genes are accentuated by the PageRank analysis and are involved in a community of human zinc finger genes indicated by the yellow edges.

4.3 18w

In Figure 5, the gene homology network for 18w in *D. melanogaster* is shown. 18w is a gene that contributes to multiple processes including ovarian follicle cell migration, antibacterial humoral response and ventral cord development (18w, Alliance of Genome Resources). The figure shows the entirety of the complex network that 18w is associated with. In Figure 6, the same network is shown, but with homology calls with scores of less than 4 removed. This demonstrates the ease with which the tool can be used to remove complexity and allow researches to focus on high value data easily.

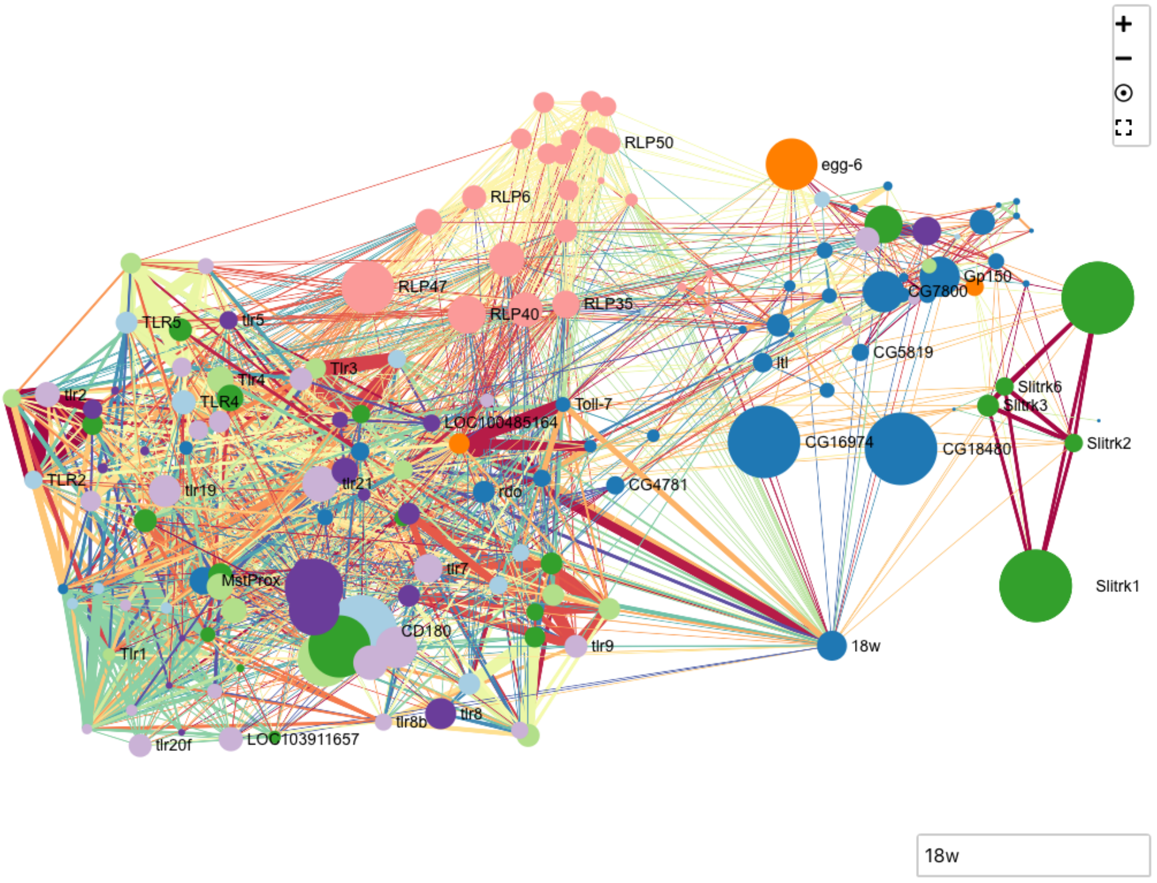


Figure 5: Gene homology network of 18w in *D. melanogaster*.

5 Discussion

Recall that the core mission behind developing the Homology Explorer application is to:

> Provide an easily accessible visual interface allowing model organism genetic researchers to intuitively explore the gene homology network.

Other key constraints of the application design include that the application be easily hostable, be built to a production-grade standard of quality, and have a reactive user interface capable of executing a variety of network analysis operations.

The Homology Explorer is a minimum viable product which accomplishes both the core mission and the constraints through a variety of means. Despite this, the application is far from perfect and, in its present state, there is more work to be done to get the application to being completely production-ready. The outcomes, limitations, and future of the Homology Explorer application are discussed in the sections below.

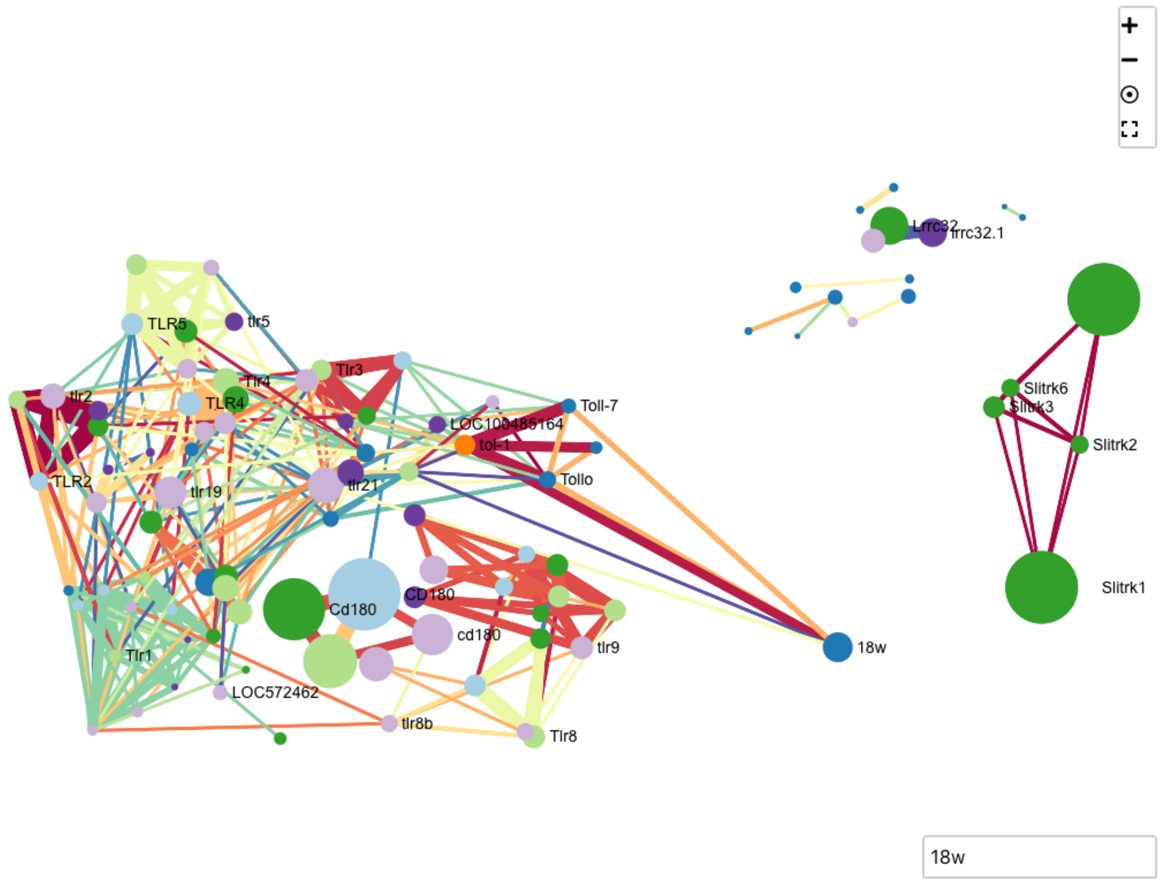


Figure 6: Gene homology network of 18w in *D. melanogaster* with homology scores of less than 4 removed.

5.1 Outcomes

Containerization of the application allows it to be run with one line of code because all of the dependencies are taken care of during the container build process. Additionally, the Proxy, API, UI services are all implemented with tools known to provide high performance and security while being scalable and also maintainable through leveraging a rich community and documentation. They were meticulously selected for this project due to their distinguishing attributes relevant to this project.

In terms of network analysis, the application distills the large gene homology graph to a network visualization of the subgraph composed of all the edges between all the nodes (genes) in the first-degree neighborhood of the requested gene or genes. Implementing this universal graph definition on all of the queries related to network discovery enables the UI to use the same API request parameterizations for separate API calls. Additionally, adding new python network analysis operations is relatively simple because the code for a new network operation component only consists of three components:

1. Instantiate an `nx.Graph` object from an edgelist, IE `[(source,target,weight),(s2,t2,w2),...]`
2. Run the network operation on the graph
3. Serialize the new attribute results to a `graphology` nodelist or edgelist, IE
   - for nodes: `{ { node: geneid, attributes: {...} }, ... }`
   - for edges: `{ { source: geneid1, target: geneid2, attributes: {...} }, ... }`

Writing a new CRUD operation following the outline above and adding it to either the node or edge analysis API endpoints is all it takes to implement a new network operation. This is a powerful characteristic because it will both streamline future development of more core network analysis features and enable users to implement their own network analyses in this web framework.

The network visualizations are fast, responsive, and intuitive for new users. There is layout flexibility to view the network in a variety of configurations that also allows for users to zoom, pan, and move nodes as they see fit. The navigational search feature in the visualization enables users to easily find the gene they are looking for and the gene info presents a nice high level summary of the gene functions. The ability to constrain the network by species enables researchers to explore paralogs, orthologs, or a mix of both, and the ability to constrain the network by edge score (the number of predictive algorithms that predict a given edge. . .) further allows researchers to focus without distraction on the most gene neighborhood network structures.

5.2 Limitations

Although the Homology Explorer accomplishes the initial project goals to a large degree, there are several limitations with the application which prevent it from being the best gene network application that it could be. Probably the largest design limitation is a result of design choices due to the large size of the entire gene homology network. Visualizing this entire network and running network analyses on it wholistically would be prohibitively computationally expensive to do so effectively in this web setting. A separate, more static visualization approach ought to be taken for a more comprehensive network-wide discovery and analysis.

The Gene Neighborhood subgraph visualization concept was implemented as a means of circumventing the issue of large network size. However, the visualization currently only visualizes the neighborhood for a single gene. Although API endpoints are in place for querying the larger neighborhood centered around multiple genes, this feature has not yet been implemented in the UI. This results in an opposite problem - where visualizing the entire network would provide far too much graph context to be useful, the single gene neighborhood may, in certain cases, be providing *too little* context to the user. This is because the current network analysis endpoints only operate on the geneneighborhood provided to them. Since this application is running the analysis on a subgraph of the gene homology network, any edges that are removed on account of one of the nodes *not* belonging to the given subgraph will bias the analysis depending on the surrounding structural features that are omitted. A consequence of this is that depending on the particular neighborhood one is observing, one might see inconsistent results between similar neighborhood analyses which share the same nodes.

Currently there are only a limited number of network analysis options available - only one community detection algorithm and one centrality algorithm. There are also *no* subgraph-wide network analysis endpoints to analyze wholistic subgraph characteristics like average degree or shortest path algorithms. Although there is some groundwork laid to develop more network analyses, this is a current shortcoming of the application in its current form.

Finally, the dynamic search for genes only returns results sorted alphabetically. This is a small but critical UI change needed for users to properly query the Homology Explorer for genes they are interested in. There are also some occasions when a queried `geneid` returns an error that it is not found. This is likely because the `geneid` is missing from the `OrthologPairs` table, but more investigation is needed to understand the root cause.

5.3 Next Steps

As identified in the limitations, there are several design improvements which could be made to improve the user experience. However, more critically important as a next step would be to implement the "multigene" neighborhood querying whereby a user can request the gene neighborhood for a list of genes instead of just a single one. This feature would allow the user a greater "line of sight" across the regional network neighborhood of the target gene or genes. Some additional nice-to-have features to accompany this might be "ballooning" the gene neighborhood subgraph by allowing the user to expand the gene neighborhood off of a selected node, and implementing n-degree neighborhood toggling, whereby the user can set the gene neighborhood consist of neighbors from either the first-degree or the nth-degree.

Ultimately, the most critical next step is to collect feedback from theand iterate over domain experts and stakeholders to determine what the priorities and future for this project should look like so as to best serve the needs of the genetic research community. For now, the Homology Explorer application is a good initial prototype for a tool which could one day provide significant utility to genetic research.

6 Acknowledgments

The authors would like to thank all of the open source developers and researchers for their efforts. The model organism databases: FlyBase, MGD, RGD, SGD, WormBase, XenBase, ZFIN, PomBase, and the Alliance of Genome Resources. We also wish to thank the National Institutes of Health (NIH) for supporting the biological data resources that were used for this project and Dr. Yanhui Hu and DIOPT for providing us with the data.

7 References

Tulpan D, Leger S. The Plant Orthology Browser: An Orthology and Gene-Order Visualizer for Plant Comparative Genomics. Plant Genome. 2017 Mar;10(1). doi: 10.3835/plantgenome2016.08.0078. PMID: 28464063.

Mustafin ZS, Lashin SA, Matushkin YG, Gunbin KV, Afonnikov DA. Orthoscape: a cytoscape application for grouping and visualization KEGG based gene networks by taxonomy and homology principles. BMC Bioinformatics. 2017 Jan 27;18(Suppl 1):1427. doi: 10.1186/s12859-016-1427-5. PMID: 28466792; PMCID: PMC5333177.

Nevers Y, Kress A, Defosset A, Ripp R, Linard B, Thompson JD, Poch O, Lecompte O. OrthoInspector 3.0: open portal for comparative genomics. Nucleic Acids Res. 2019 Jan 8;47(D1):D411-D418. doi: 10.1093/nar/gky1068. PMID: 30380106; PMCID: PMC6323921.

Hu Y, Flockhart I, Vinayagam A, Bergwitz C, Berger B, Perrimon N, Mohr SE. An integrative approach to ortholog prediction for disease-focused and other functional studies. BMC Bioinformatics. 2011 Aug 31;12:357. doi: 10.1186/1471-2105-12-357. PMID: 21880147; PMCID: PMC3179972.

PTEN, https://www.alliancegenome.org/gene/HGNC:9588#disease-associations

BCL6, https://www.alliancegenome.org/gene/HGNC:1001#disease-associations

18w, https://www.alliancegenome.org/gene/FB:FBgn0004364#summary

Koonin EV. Orthologs, paralogs, and evolutionary genomics. Annu Rev Genet. 2005;39:309-38. doi: 10.1146/annurev.genet.39.073003.114725. PMID: 16285863.

Millburn GH, Crosby MA, Gramates LS, Tweedie S; FlyBase Consortium. FlyBase portals to human disease research using Drosophila models. Dis Model Mech. 2016 Mar;9(3):245-52. doi: 10.1242/dmm.023317. PMID: 26935103; PMCID: PMC4826978.

Bulaklak K, Gersbach CA. The once and future gene therapy. Nat Commun. 2020 Nov 16;11(1):5820. doi: 10.1038/s41467-020-19505-2. PMID: 33199717; PMCID: PMC7670458.